Data Abstraction

Essentials of Programming Languages (Chapter 2)

I appreciate Dr. Shirin Baghoolizadeh for her helps for preparing this presentation.

Specifying Data via Interfaces

- We do not want to be concerned with representations of data types
  - manipulation files
  - arithmetic operations...
- We may also decide to change the representation of the data
- Data abstraction divides a data type into:
  - an interface: It tells us
    - what the data of the type represents
    - what the operations on the data are
    - what properties these operations may be relied on to have
  - an implementation: It provides
    - a specific representation of the data
    - code for the operations that make use of that data representation

Representation

$\lceil v \rceil$: The representation of data v

- Data type of natural numbers:
  - The interface is to consist of four procedures: zero, is-zero?, successor, and predecessor

$$\texttt{(zero)} = \lceil 0 \rceil$$

$$\texttt{(is-zero?}\ \lceil n \rceil\texttt{)} = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$\texttt{(successor}\ \lceil n \rceil\texttt{)} = \lceil n+1 \rceil \quad (n \geq 0)$$

$$\texttt{(predecessor}\ \lceil n+1 \rceil\texttt{)} = \lceil n \rceil \quad (n \geq 0)$$

Client Program

- The client code manipulates the new data only through the operations specified in the interface
  - This code is representation-independent
- A sample of client program, manipulating natural numbers:

```
(define plus
  (lambda (x y)
    (if (is-zero? x)
      y
      (successor (plus (predecessor x) y)))))
```

No matter what representation is in use

Interfaces

- Most interfaces contain
  - some *constructors*
    - for building elements of the data type
  - some *observers*
    - for extract information from values of the data type

- *Zero*, *successor*, and *predecessor* are constructors and *is-zero?* is observer

Representation

Some representations for natural numbers:

1) Unary representation:

- natural number *n* is represented by a list of n #t's
- 0 is represented by (), 1 is represented by (#t), 2 is represented by (#t #t), ...

$$\lceil 0 \rceil = \texttt{()}$$
$$\lceil n+1 \rceil = \texttt{(\#t . } \lceil n \rceil \texttt{)}$$

In this representation, we can satisfy the specification by writing

```
(define zero (lambda () '()))
(define is-zero? (lambda (n) (null? n)))
(define successor (lambda (n) (cons #t n)))
(define predecessor (lambda (n) (cdr n)))
```

Representation

2) Scheme number representation:

- we simply use Scheme’s internal representation of numbers

```
(define zero (lambda () 0))
(define is-zero? (lambda (n) (zero? n)))
(define successor (lambda (n) (+ n 1)))
(define predecessor (lambda (n) (- n 1)))
```

Representation

3) Bignum representation:

- Numbers are represented in base N, for some large integer N
- The representation becomes a list consisting of numbers between 0 and N −1
  - with least-significant bigit first
- This representation makes it easy to represent integers that are much larger than can be represented in a machine word

$$\lceil n \rceil = \begin{cases} () & n = 0 \\ (r \;\; . \;\; \lceil q \rceil) & n = qN + r,\ 0 \leq r < N \end{cases}$$

So if $N = 16$, then $\lceil 33 \rceil =$ `(1 2)` and $\lceil 258 \rceil =$ `(2 0 1)`, since

$$258 = 2 \times 16^0 + 0 \times 16^1 + 1 \times 16^2$$

Opaque vs. Transparent

- Types can be divided into:
  - Opaque types
    - The representation of a type is hidden, so it cannot be exposed by any operation (including printing)
  - Transparent types
- Scheme does not provide a standard mechanism for creating new opaque types
  - we define interfaces and rely on the writer of the client program to be discreet and use only the procedures in the interfaces

Representation Strategies for Data Types

Consider a data type of *environments*.

- An environment associates a value with each element of a finite set of variables
- An environment is a function whose domain is a finite set of variables, and whose range is the set of all Scheme values
- environment *env* = $\{(var_1, val_1), \ldots, (var_n, val_n)\}$

The interface to this data type has three procedures, specified as follows:

$$
\begin{aligned}
&(\texttt{empty-env}) && = \lceil \emptyset \rceil \\
&(\texttt{apply-env}\ \lceil f \rceil\ \ var) && = f(var) \\
&(\texttt{extend-env}\ \ var\ \ v\ \ \lceil f \rceil) && = \lceil g \rceil, \\
& && \text{where } g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases}
\end{aligned}
$$

Environment

For example, the expression

```
> (define e
    (extend-env 'd 6
      (extend-env 'y 8
        (extend-env 'x 7
          (extend-env 'y 14
            (empty-env))))))
```

defines an environment e such that $e(\mathtt{d}) = 6$, $e(\mathtt{x}) = 7$, $e(\mathtt{y}) = 8$, and e is undefined on any other variables.

In this example, `empty-env` and `extend-env` are the constructors, and `apply-env` is the only observer

Environment

We can obtain a representation of environments by observing that every environment can be built by starting with the empty environment and applying `extend-env` n times, for some $n \geq 0$, e.g.,

```
(extend-env varₙ valₙ
   ...
   (extend-env var₁ val₁
     (empty-env))...)
```

So every environment can be built by an expression in the following grammar:

$$
\begin{aligned}
\textit{Env-exp} &::= \texttt{(empty-env)} \\
&::= \texttt{(extend-env } \textit{Identifier} \ \ \textit{Scheme-value} \ \ \textit{Env-exp}\texttt{)}
\end{aligned}
$$

A data-structure representation of environments

$$
\begin{aligned}
Env &= \texttt{(empty-env)} \mid \texttt{(extend-env } Var\ SchemeVal\ Env\texttt{)} \\
Var &= Sym
\end{aligned}
$$

empty-env : $() \rightarrow Env$

```
(define empty-env
  (lambda () (list 'empty-env)))
```

extend-env : $Var \times SchemeVal \times Env \rightarrow Env$

```
(define extend-env
  (lambda (var val env)
    (list 'extend-env var val env)))
```

A data-structure representation of environments

apply-env : $Env \times Var \rightarrow SchemeVal$

```
(define apply-env
  (lambda (env search-var)
    (cond
      ((eqv? (car env) 'empty-env)
       (report-no-binding-found search-var))
      ((eqv? (car env) 'extend-env)
       (let ((saved-var (cadr env))
             (saved-val (caddr env))
             (saved-env (cadddr env)))
         (if (eqv? search-var saved-var)
           saved-val
           (apply-env saved-env search-var))))
      (else
        (report-invalid-env env)))))

(define report-no-binding-found
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))

(define report-invalid-env
  (lambda (env)
    (eopl:error 'apply-env "Bad environment: ~s" env)))
```

Env = (empty-env) | (extend-env Var $SchemeVal$ Env)

Procedural Representation

$Env = Var \to SchemeVal$

empty-env : $() \to Env$

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (report-no-binding-found search-var))))
```

extend-env : $Var \times SchemeVal \times Env \to Env$

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var)))))
```

apply-env : $Env \times Var \to SchemeVal$

```
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

apply-env : $Env \times Var \to SchemeVal$

```
(define apply-env
  (lambda (env search-var)
    (cond
      ((eqv? (car env) 'empty-env)
       (report-no-binding-found search-var))
      ((eqv? (car env) 'extend-env)
       (let ((saved-var (cadr env))
             (saved-val (caddr env))
             (saved-env (cadddr env)))
         (if (eqv? search-var saved-var)
           saved-val
           (apply-env saved-env search-var))))
      (else
        (report-invalid-env env)))))

(define report-no-binding-found
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))

(define report-invalid-env
  (lambda (env)
    (eopl:error 'apply-env "Bad environment: ~s" env)))
```

Interfaces for Recursive Data Types

$$
\begin{aligned}
\mathit{Lc\text{-}exp} &::= \mathit{Identifier} \\
&::= (\texttt{lambda}\ (\mathit{Identifier})\ \mathit{Lc\text{-}exp}) \\
&::= (\mathit{Lc\text{-}exp}\ \ \mathit{Lc\text{-}exp})
\end{aligned}
$$

The definition of occurs-free? in section 1.2.4 is not as readable as it might be

Our interface will have constructors and two kinds of observers: predicates and extractors

An Interface for Lambda calculus expressions

$$
\begin{aligned}
\textit{Lc-exp} &::= \textit{Identifier} \\
&::= (\texttt{lambda}\ (\textit{Identifier})\ \ \textit{Lc-exp}) \\
&::= (\textit{Lc-exp}\ \ \textit{Lc-exp})
\end{aligned}
$$

The constructors are:

var-exp	$: \mathit{Var} \rightarrow \textit{Lc-exp}$
lambda-exp	$: \mathit{Var} \times \textit{Lc-exp} \rightarrow \textit{Lc-exp}$
app-exp	$: \textit{Lc-exp} \times \textit{Lc-exp} \rightarrow \textit{Lc-exp}$

The predicates are:

var-exp?	$: \textit{Lc-exp} \rightarrow \mathit{Bool}$
lambda-exp?	$: \textit{Lc-exp} \rightarrow \mathit{Bool}$
app-exp?	$: \textit{Lc-exp} \rightarrow \mathit{Bool}$

Finally, the extractors are

var-exp->var	$: \textit{Lc-exp} \longrightarrow \mathit{Var}$
lambda-exp->bound-var	$: \textit{Lc-exp} \longrightarrow \mathit{Var}$
lambda-exp->body	$: \textit{Lc-exp} \longrightarrow \textit{Lc-exp}$
app-exp->rator	$: \textit{Lc-exp} \longrightarrow \textit{Lc-exp}$
app-exp->rand	$: \textit{Lc-exp} \longrightarrow \textit{Lc-exp}$

Occurs-free?

$$Lc\text{-}exp ::= Identifier$$
$$::= (\texttt{lambda}\ (Identifier)\ \ Lc\text{-}exp)$$
$$::= (Lc\text{-}exp\ \ Lc\text{-}exp)$$

occurs-free? : $Sym \times LcExp \to Bool$

```
(define occurs-free?
  (lambda (search-var exp)
    (cond
      ((var-exp? exp) (eqv? search-var (var-exp->var exp)))
      ((lambda-exp? exp)
       (and
         (not (eqv? search-var (lambda-exp->bound-var exp)))
         (occurs-free? search-var (lambda-exp->body exp))))
      (else
        (or
          (occurs-free? search-var (app-exp->rator exp))
          (occurs-free? search-var (app-exp->rand exp)))))))
```

Interfaces for Recursive Data Types

Designing an interface for a recursive data type

1. *Include one constructor for each kind of data in the data type.*
2. *Include one predicate for each kind of data in the data type.*
3. *Include one extractor for each piece of data passed to a constructor of the data type.*

A Tool for Defining Recursive Data Types

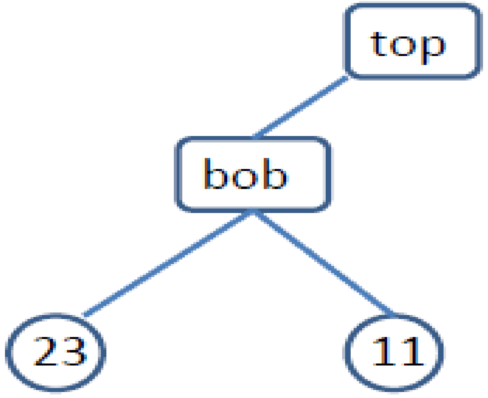
```
(require (lib "eopl.ss"  "eopl"))
```

(define-datatype *type-name* *type-predicate-name*
  $\{$ (*variant-name* $\{$ (*field-name* *predicate*) $\}^*$) $\}^+$)

```
(define-datatype binaryTree binaryTree?
          (null-node)
          (leaf-node (datum number?))
          (interior-node (key symbol?)
                         (left-child binaryTree?)
                         (right-child binaryTree?)))
```

A Tool for Defining Recursive Data Types

```
(define T1 (leaf-node 23))
(define T2 (leaf-node 11))
(define T3 (interior-node 'bob T1 T2))
(define T4 (interior-node 'top T3 (null-node)))
```



Lc-exp (define-datatype)

```
(define-datatype lc-exp lc-exp?
  (var-exp
    (var identifier?))
  (lambda-exp
    (bound-var identifier?)
    (body lc-exp?))
  (app-exp
    (rator lc-exp?)
    (rand lc-exp?)))
```

$$
\begin{aligned}
\mathit{Lc\text{-}exp} &::= \mathit{Identifier} \\
&::= (\texttt{lambda}\ (\mathit{Identifier})\ \mathit{Lc\text{-}exp}) \\
&::= (\mathit{Lc\text{-}exp}\ \mathit{Lc\text{-}exp})
\end{aligned}
$$

The constructors are:

var-exp	: $Var \to Lc\text{-}exp$
lambda-exp	: $Var \times Lc\text{-}exp \to Lc\text{-}exp$
app-exp	: $Lc\text{-}exp \times Lc\text{-}exp \to Lc\text{-}exp$

The predicates are:

var-exp?	: $Lc\text{-}exp \to Bool$
lambda-exp?	: $Lc\text{-}exp \to Bool$
app-exp?	: $Lc\text{-}exp \to Bool$

var-exp->var	: $Lc\text{-}exp \to Var$
lambda-exp->bound-var	: $Lc\text{-}exp \to Var$
lambda-exp->body	: $Lc\text{-}exp \to Lc\text{-}exp$
app-exp->rator	: $Lc\text{-}exp \to Lc\text{-}exp$
app-exp->rand	: $Lc\text{-}exp \to Lc\text{-}exp$

S-list (define-datatype)

```
(define-datatype s-list s-list?
  (empty-s-list)
  (non-empty-s-list
    (first s-exp?)
    (rest s-list?)))

(define-datatype s-exp s-exp?
  (symbol-s-exp
    (sym symbol?))
  (s-list-s-exp
    (slst s-list?)))
```

$$\begin{aligned} \mathit{S\text{-}list} &::= (\{\mathit{S\text{-}exp}\}^*) \\ \mathit{S\text{-}exp} &::= \mathit{Symbol} \mid \mathit{S\text{-}list} \end{aligned}$$

Cases

There is also a cases expression that gives you access to the variant fields of an object constructed with define-datatype.

The format of this is

```
(cases  type  object
        (variant1 (data fields) exp1)
        (variant2 (data fields) exp2)
        etc. )
```

Cases

```
(define sum (lambda (tree)
    (cases binaryTree tree
        (null-node () 0)
        (leaf-node (v) v)
        (interior-node (sym left right) (+ (sum left) (sum right))))))
```

Occurs-free (cases)

occurs-free? : $Sym \times LcExp \to Bool$

```
(define occurs-free?
  (lambda (search-var exp)
    (cases lc-exp exp
      (var-exp (var) (eqv? var search-var))
      (lambda-exp (bound-var body)
        (and
          (not (eqv? search-var bound-var))
          (occurs-free? search-var body)))
      (app-exp (rator rand)
        (or
          (occurs-free? search-var rator)
          (occurs-free? search-var rand))))))
```

Domain-specific language

- The form define-datatype is an example of a *domain-specific language*
- A domain-specific language is a small language for describing a single task
- In this case, the task was defining a recursive data type
- Such a language may lie inside a general-purpose language, as define-datatype does, or it may be a standalone language

Abstract Syntax and Its Representation

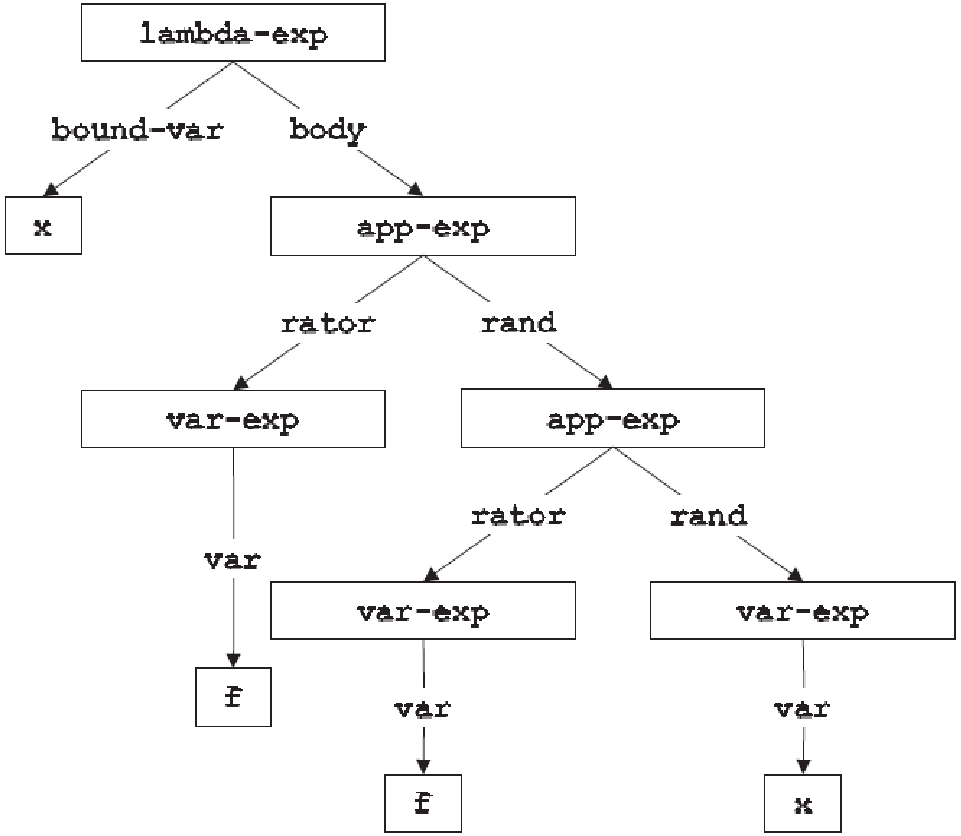
- A grammar usually specifies a particular representation of an inductive data type
  - Such a representation is called concrete syntax, or external representation

$$
\begin{aligned}
\mathit{Lc\text{-}exp} &::= \mathit{Identifier} \\
&::= \texttt{proc}\ \ \mathit{Identifier}\ \ \texttt{=>}\ \ \mathit{Lc\text{-}exp} \\
&::= \mathit{Lc\text{-}exp}\ (\mathit{Lc\text{-}exp})
\end{aligned}
$$

- To process such data, we need to convert it to an internal representation
  - The define-datatype form provides a convenient way of defining such an internal representation.
  - We call this abstract syntax
  - Terminals such as parentheses need not be stored, because they convey no information

Abstract syntax tree

It is convenient to visualize the internal representation as an abstract syntax tree




```
(lambda (x) (f (f x)))
```

Abstract Syntax Tree

- Each internal node of the tree is labeled with the associated production name
- Edges are labeled with the name of the corresponding nonterminal occurrence
- Leaves correspond to terminal strings

$$
\begin{aligned}
\mathit{Lc\text{-}exp} &::= \mathit{Identifier} \\
&\quad \boxed{\texttt{var-exp (var)}} \\
&::= (\texttt{lambda}\ (\mathit{Identifier})\ \mathit{Lc\text{-}exp}) \\
&\quad \boxed{\texttt{lambda-exp (bound-var body)}} \\
&::= (\mathit{Lc\text{-}exp}\ \mathit{Lc\text{-}exp}) \\
&\quad \boxed{\texttt{app-exp (rator rand)}}
\end{aligned}
$$

Create an Abstract Syntax

- To create an abstract syntax for a given concrete syntax, we must name each production of the concrete syntax and each occurrence of a nonterminal in each production
- It is straightforward to generate define-datatype declarations for the abstract syntax. We create one define-datatype for each nonterminal, with one variant for each production.

Parsing

- If the concrete syntax is a set of strings of characters, it may be a complex undertaking to derive the corresponding abstract syntax tree
- This task is called parsing and is performed by a parser
- Because writing a parser is difficult in general, it is best performed by a tool called a parser *generator*
- A parser generator takes as input a grammar and produces a parser

Converting a Concrete Syntax to Abstract Syntax

If the concrete syntax is given as a set of lists, the parsing process is considerably simplified

parse-expression : *SchemeVal* → *LcExp*

```
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
       (if (eqv? (car datum) 'lambda)
         (lambda-exp
           (car (cadr datum))
           (parse-expression (caddr datum)))
         (app-exp
           (parse-expression (car datum))
           (parse-expression (cadr datum)))))
      (else (report-invalid-concrete-syntax datum)))))
```

Converting an Abstract Syntax to Concrete Syntax

It is usually straightforward to convert an abstract syntax tree back to a list-and-symbol representation. If we do this, the Scheme print routines will then display it in a list-based concrete syntax. This is performed by `unparse-lc-exp`:

unparse-lc-exp : $LcExp \to SchemeVal$

```
(define unparse-lc-exp
  (lambda (exp)
    (cases lc-exp exp
      (var-exp (var) var)
      (lambda-exp (bound-var body)
        (list 'lambda (list bound-var)
          (unparse-lc-exp body)))
      (app-exp (rator rand)
        (list
```

Interpreter Recipe

The Interpreter Recipe

1. *Look at a piece of data.*
2. *Decide what kind of data it represents.*
3. *Extract the components of the datum and do the right thing with them.*